JGOMAS user manual

GTI-IA (DSIC - UPV)

1 Introduction

This document presents social simulator Multi-Agent System (MAS) based on JADE that has been developed to fulfill different purposes. The main one is to serve as an initial point for a feasibility study of the integration of MAS and Virtual Reality (VR). In this way, it combines a 3D graphic viewer with a MAS that can be used in a qualitative evaluation of the current performance of the MAS.

This way, have arisen *JGOMAS* (Game Oriented Multi-Agent System, based on Jade) as a test platform to study a full integration between MAS and VR, allowing also to be used as a validation toolkit for MAS systems.

As the option for the environment to simulate in the developed MAS, it has been elected a capture-the-flag (CTF) kind of game.

1.1 Capture the Flag

In this kind of games, two teams (red and blue, allies and axis) must compete to capture the opponent's flag. This game modality has become an standard included in almost all multiplayer games appeared since Quake [1].

It is very easy and intuitive to apply *Multi-Agent Systems* to this type of games, because each soldier may be seen as an agent. Moreover, agents in a team must cooperate among them to get the team's objective. In this way, they compete with the other team.

So, a CTF game is proposed as the kind of social interaction to simulate, where the agents group in two teams (allies and axis). On one hand, allies agents must go to axis base, capture the flag and take it to their base, in which case allied team win the game. On the other hand, axis agents defend their flag against the other team and, if the flag is captured, they must return it to their base. There is a limit time for allies to bring the flag to their base. If time expires, axis team win the game.

Of course, it is necessary an additional module which will display the 3D virtual environment: agents, objects and scenario.

2 JGOMAS Description

This section details the system architecture being developed to solve the problems above described. This system *JGOMAS*, is a platform to execute a *Multi-Agent System* in a 3D virtual environment. Basically, a handful of agents spreaded out over two teams, with objectives to carry out, and, of course, integrated in a virtual environment.

This platform can be used as simulator for validation of coordination, comunication and learning algorithms in the field of *Multi-Agent Systems* (or *Artificial Inteligence* in general). To get this, *JGOMAS* must allow the user to add his own code modifications (*mods*). Furthermore, *JGOMAS* can be used to study a full integration of *Multi-Agent Systems* and *Virtual Reality Systems*. In this way, results can be analyzed either quantitative or qualitative.

Finally, *JGOMAS* must be multiplatform, besides to respect standards in the field of *Multi-Agent Systems* (for example, FIPA [6]).

2.1 Architecture

JGOMAS is composed mainly of two subsystems. On one hand, there is a *Multi-Agent System* with two different kind of agents running on. One of these agents will control the current game logic, whilst the others belong to one of the teams, and they will be playing the entire game. Really, this subsystem is an upper layer over a existing *Multi-Agent System* platform, specifically JADE [4], so it can take advantage of all services that JADE provides. Due to the usage of JADE as support layer for *JGOMAS*, agents must be written in JAVA [5].

On the other hand, it has been developed an *ad-hoc* graphic viewer (*Render Engine*) to display a 3D virtual environment. According to requirements of graphic applications (high computational cost for short periods), this *Render Engine* has been designed as an external module (and not as an agent). It has been written in C++, using the graphic library OpenGL [2].

Figure 1 shows an overview of *JGOMAS* architecture, where all its components and its relationships can be seen: JADE platform as support of *JGOMAS* MAS, which is composed of agents, one of them acting as controller for the rest of agents, and as interface for graphic viewer application.

JGOMAS MAS can be seen as a kernel (basic package), which provides an interface for Render Engine to establish a connection to the current game.

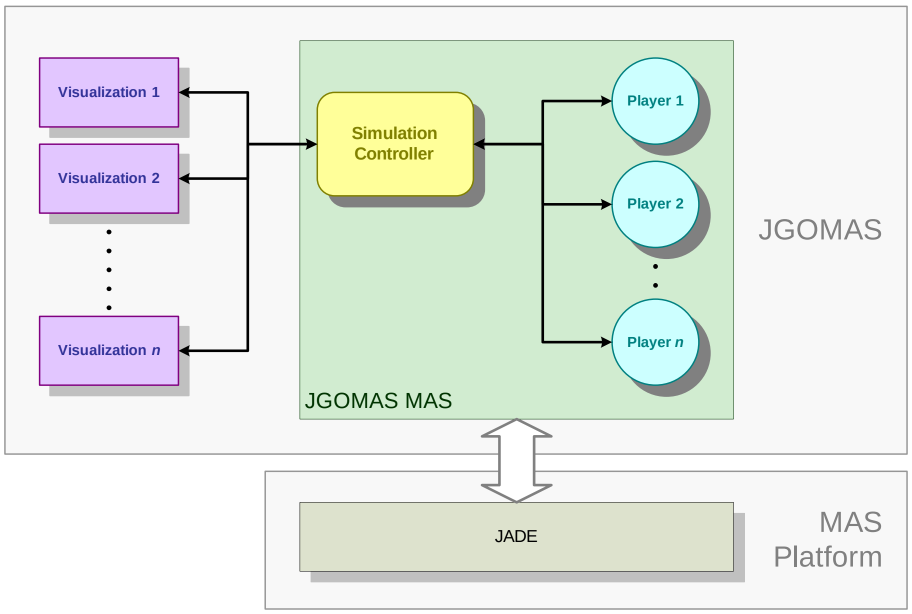


Figure 1: *JGOMAS* architecture overview

2.2 JGOMAS MAS

As it has been commented, *JGOMAS* MAS works over JADE platform. JADE (Java Agent DEvelopment Framework) is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications.

2.2.1 JGOMAS Agent Taxonomy

JGOMAS has defined a class inheritance hierarchy for agents. At the top of the agent hierarchy, there is the base class *JGomasAgent*. It inherits directly from JADE's *Agent* class, and it provides a set of basic features/services to all *JGOMAS* agents (e.g. services registry). So, *JGOMAS* agents must derive from *JGomasAgent* class. Moreover, a *JGOMAS* agent can be, according to its modifiability, an internal or external one.

Internal agents: those who are staff in the *JGOMAS* MAS subsystem. Its behaviors are predefined, and user cannot change them. An agent must specialize in:

Manager: this is an special agent. Its main goal is to coordinate the current game. Besides, it must answer to requests of the rest of agents. Another task it does is to provide an interface for Render Engine. Thus, any instance of Render Engine can connect to the current game to display the 3D virtual environment. Due to the *Manager Agent* importance, the next section do a more detailed explanation of this agent.

Pack: those are *medic packs* (used to give health to the agents), *ammo packs* (used to give ammunition to the agents) and the *objective pack*, that is, the flag to capture. They are created and destroyed dynamically during the current game, with the exception of *objective pack* (there is only one flag, and it exists during all the game and can not be destroyed).

External agents: they are really the players of the current game. They have a set of basic predefined behaviors. However, user can modify those behaviors or even add new ones.

BasicTroop: user agents, where each one is performing a role. Each role has different features, services and behaviors. Furthermore, an agent can play a unique role during the current game. There are defined three roles (but user can define new ones), each one providing a unique service. So *troop* agents are specialized in:

- *Soldier*: provides a *CallForBackup* service (agent goes to help teammates).
- *Medic*: provides a *CallForMedic* service (agent goes to give medic packs).
- *FieldOps*: provides a *CallForAmmo* service (agent goes to give ammo packs).

Figure 2 shows the *JGOMAS* taxonomy tree, where the bottom level is the most specialized. Having in mind that the number of agents is limited, user's election of roles is a decisive factor to win the game.

2.3 Agent Manager

This is an special agent in the *JGOMAS* MAS subsystem. In fact, there is only one instance of it running during the current game. It has to do two very different tasks:

- Interface for Graphic Viewer.
- Game Logic Management.

2.3.1 Interface for Graphic Viewer

Agent Manager is in charge of functioning as a server for any Graphic Viewer client interested in connecting to the current game. At the beginning

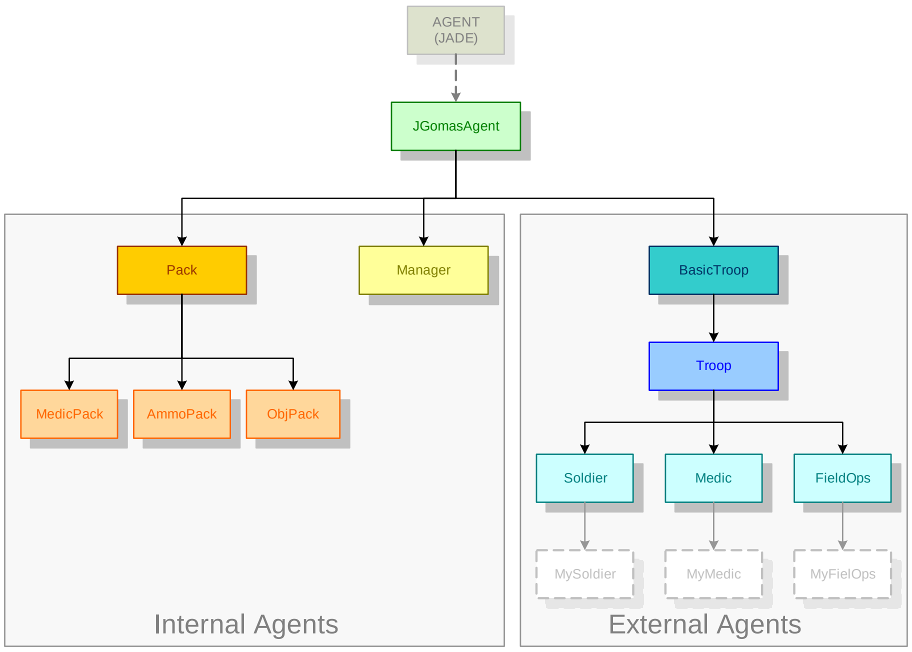


Figure 2: *JGOMAS* Agent Taxonomy. The bottom level is the most specialized

of *JGOMAS*, *Agent Manager* executes a thread. This thread is simply a server for Graphic Viewer clients. First, the server creates a listening socket in a specific port, and waits for connection attempts from clients. The client creates a socket on its side, and attempts to connect with the server. The server then accepts the connection, and communication can begin. For each accepted connection, a new thread is executed. So, we can have several viewers running at the same time (perhaps in different machines), all connected to the current game.

Moreover, *Agent Manager* holds the game state: *troop* agents, static and dynamic objects and their main attributes (position, direction, velocity, etc.). So, it sends all this information to each graphic viewer client connected, once for frame. Thus, graphic viewers can render their images continuously, keeping the desired framerate. Figure 3 shows *JGOMAS* architecture, and how agents, JADE platform and graphic viewers are integrated.

2.3.2 Game Logic Management

This subject is really a level of abstraction over the *JGOMAS* MAS. Game Logic involves many aspects, but all oriented to manage the course of the current game. For example, *When does the match start? Which is the map to play? Where are agents and what are they seeing at a certain*

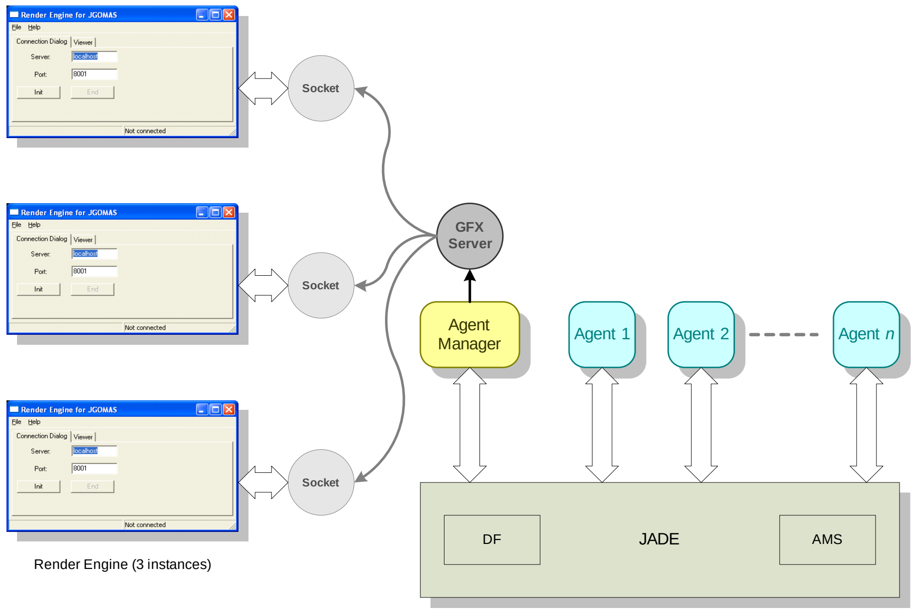


Figure 3: *JGOMAS* architecture and integration of its subsystems

point? and so on...

The control of game logic is centralized just in one place: the *Manager* agent. It is the agent in charge of some tasks, as:

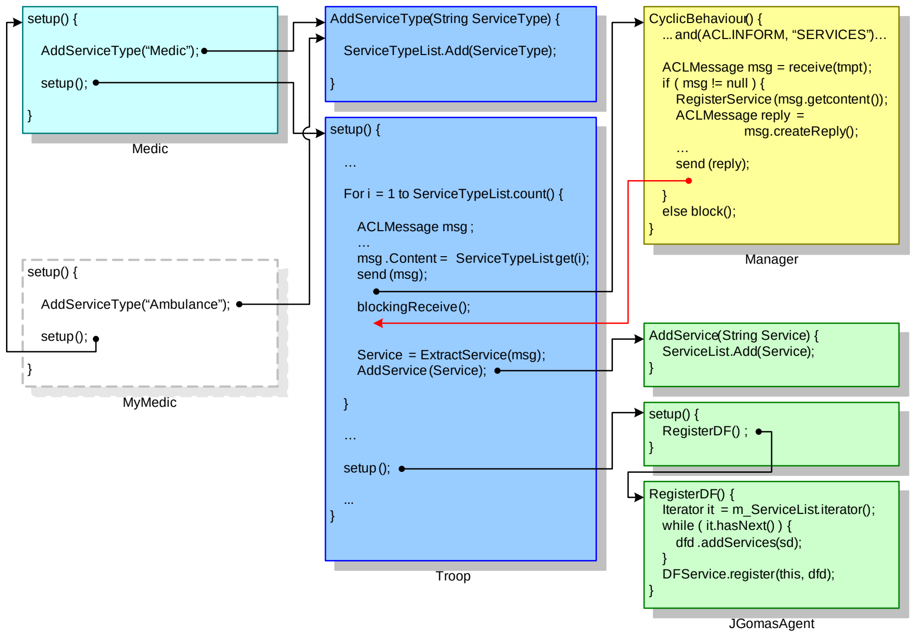
- Management of the life-cycle of the current game: it is in charge of synchronization of all agents for the beginning of the game, and their destruction at the end of the game, besides other more specific details of the game control like inform about the match's map, the objective, etc.

- Coordination and management of services registration: an agent cannot register a service if *Agent Manager* doesn't allow him to. Moreover, it can manipulate the service's name to prevent cheating (agents from one team subscribing to services of the other team). This is showed in the execution trace at figure 4.

- Holding the game state of the current game: Each agent calculates its new position and action to do. Then, they send all that information to *Agent Manager*. So, it controls all information regarding the current game state, agents and their main attributes (position, direction, velocity, etc.).

- Attention of some agent's requests regarding interactions with the environment: *Manager* listen to requests, process them, and returns the results to agents requesting information regarding actions such as look or shot.

- Statistics about agents efficiency: the *Agent Manager* is also in charge of calculating a report about the development of the current game. The purpose of this report is to have a quantitative measure to complement the qualitative data offered by the Graphic Viewer.



```
setup() {
   AddServiceType("Medic");
   setup();
}
```

```
setup() {
   AddServiceType("Ambulance");
   setup();
}
```

```
AddServiceType(String ServiceType) {
   ServiceTypeList.Add(ServiceType);
}
```

```
setup() {
   ...
   For i = 1 to ServiceTypeList.count() {
      ACLMessage msg ;
      ...
      msg .Content =  ServiceTypeList.get(i);
      send (msg);

      blockingReceive();

      Service  = ExtractService(msg);
      AddService (Service);
   }
   ...
   setup ();
   ...
}
```

```
CyclicBehaviour() {
   ... and(ACL.INFORM, "SERVICES")...

   ACLMessage msg = receive(tmpt);
   if ( msg != null ) {
      RegisterService (msg.getcontent());
      ACLMessage reply  =
                     msg.createReply();
      ...
      send (reply);

   }
   else block();
}
```

```
AddService(String Service) {
   ServiceList.Add(Service);
}
```

```
setup() {
   RegisterDF() ;
}
```

```
RegisterDF() {
   Iterator it  = m_ServiceList.iterator();
   while ( it.hasNext() ) {
      dfd .addServices(sd);
   }
   DFService.register(this, dfd);
}
```

Figure 4: Execution trace of a service registration by a *MyMedic* agent.

2.4 Render Engine

Render Engine is the graphic viewer application developed to display the 3D virtual environment in *JGOMAS*. This virtual environment includes agents, objects and scenario. This application is an important part of *JGOMAS* system. However, it is not essential, because another graphic engine (both commercial and open source) could be used to get better image rendering. To get this, it has just to be added the communication layer (connection to *Agent Manager*, communication protocol, etc.) to the new graphic engine. Normally, these engines are ready to handle a scene graph, meshes, geometry, textures, and so on. So we can take advantage of these engines, avoiding unnecessary effort.

This allows *JGOMAS* to be independent of any graphic subject (that is, visual results). This way, user could apply other concepts of *Virtual*

Reality Systems, for example, *JGOMAS* could be viewed stereoscopically, if graphic engine supports it.

The current used Render Engine is written in C++ using the OpenSceneGraph (OSG) toolkit [3]. The reason is that graphic applications, and specially *Virtual Reality Systems*, have a high computational cost. Moreover, OSG uses OpenGL standard. This ensures compatibility amongst graphic card manufacturers.

On the other hand, due to the purpose of having *JGOMAS* as a multiplatform system, Render Engine must also be multi-platform. It was one of the reasons to use for its development OSG, because OpenSceneGraph is an OpenSource, cross platform graphics toolkit for the development of high performance graphics applications. Figure 5 shows the structure of the Render Engine's architecture.

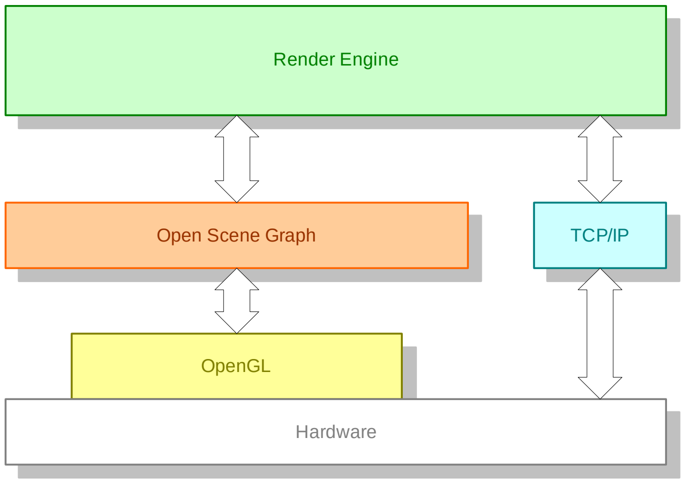


Figure 5: Render Engine's multi-layer architecture

3 Using JGOMAS

3.1 Implementation

JGOMAS uses JADE as MAS platform to take advantage of all resources it offers: behaviour mechanisms, message passing (where FIPA ACL is the language to represent messages), naming service and yellow-page service, FIPA interaction protocols, etc., in compliance with the FIPA specifications.

JGOMAS agents are written in JAVA to make the most of JADE's features. Thus, they are FIPA compliant, besides platform-independent.

3.2 Code Modifications

A user may configure *JGOMAS* MAS to his needs, and improve the *JGOMAS* agents intelligence through an API, a set of basic services, behaviors and methods, *JGOMAS* kernel offers.

User can add new source code to develop his new agents. This new source code (mods) will be integrated into the *JGOMAS* kernel at runtime. This allows the user:

- To create new roles (specialized roles) derived from *Troop* class, or any of its inherited roles (i. e., *Soldier*, *Medic* and *FieldOps*). Thus, *JGOMAS* taxonomy is extended.
- To provide new services, or to modify existing ones.
- To add new behaviors to launch a new strategy to get the objective.
- To add new features and functionality to take complex decisions which will influence in both team and individual emergent behavior. To use the *JGOMAS* API to do this, it has to be taken into account the agent working cycle (implemented as a Finite State Machine –FSM–).
- To improve path generation.
- Etc.

3.2.1 Finite State Machine.

To obtain a customizable architecture that may accept user code, agents have to have at one's disposal a generic working mechanism. This mechanism have to be able to solve automatically different kind of tasks. The chosen mechanism is implemented as a FSM, formed by three states (figure 6):

STANDING: is the initial state. When an agent comes to this state, it extracts the most priority task from the list of pending tasks. Next, agent goes to state *GOTO TARGET*.

GOTO TARGET: once an agent knows which one is the current task, it keeps in this state till it arrives to the place where it has to carry out the task. Then, agent goes to state *TARGET REACHED*.

TARGET REACHED: in this state, an agent carries out the current task. When it has finished it, agent erases it from the list of pending tasks, and agent goes to state *STANDING*, ready to get other task.

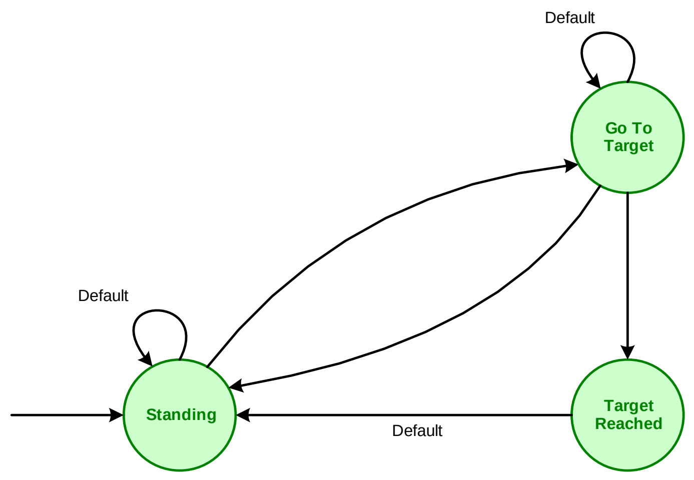


Figure 6: FSM used to handle the working cycle of a *JGOMAS* agent.

User can create new classes of agents, derived from existing ones. This way, user can either overload some methods (predefined in the kernel) and add new ones. It is very useful to download the file(s) of example of source code. Concretly, there is a file which is a basic skeleton for a new class (MyMedic) derived from CMedic class. Some interesting methods to overload are:

- protected void UpdateTargets(); ⇒ It may be used to update priority of all 'prepared (to execute)' (or pending) tasks.

- protected boolean ShouldUpdateTargets(); ⇒ When an agent is in the state **GOTO TARGET**, it can go to the state **STANDING** to recalculate the priority of his pending tasks.

- protected boolean GetAgentToAim(); ⇒ It calculates if there is an enemy at sight.

- protected void PerformLookAction(); ⇒ Action to perform when the agent is looking at, according to the objects or agents there are in the *Field of View* (FOV).

- protected boolean checkMedicAction(); ⇒ It decides, when the agent receives a Call For Medic request, if it accepts the proposal.

These methods are executed normally in each working cicle of the agent.

In this way, and as it has been mentioned before, *JGOMAS* can be used as a testbed for proofs and validation of AI algorithms.

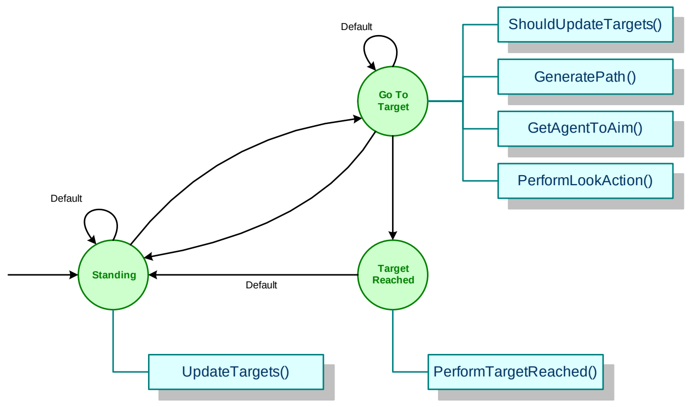


Figure 7: Some interesting methods to overload in the working cicle of an agent.

3.3 User evaluation

Finally, another important subject is the game's evaluation. User can evaluate what happened during the game, both in a quantitative and in a qualitative ways.

On one hand, user can make a qualitative evaluation because there is a graphical component which allows user to see the game's evolution. For example, user can check if an agent moves correctly with his new path generation algorithm, or if the strategic distribution of agents in the map is as he designed.

On the other hand, quantitative evaluation refers to statistics generated at the end of the match. Thus, user can check agents' efficiency. For example, the number of *medic packs* delivered versus which ones were picked up by team-mates, or by enemies. This allows user to have a numerical result to compare it against other matches played.

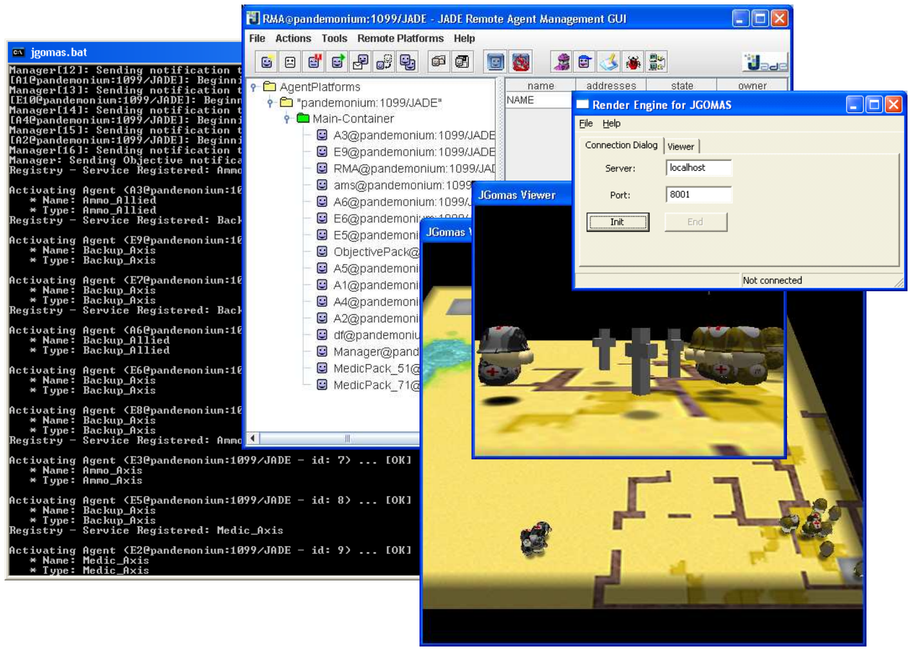
4 Launching JGOMAS

User can launch *JGOMAS* (all agents) from JADE GUI, one by one. This can be hard work, and due to *Agent Manager* needs all agents be connected to begin the current game, some scripts are given. In this way, although user executes Render Engine, it will not display agents if the current game has not begun.

The *JGOMAS* package is available for downloading from

<http://www.dsic.upv.es/users/ia/sma/tools/jgomas/>)

includes the platform, Render Engine, maps, documentation and a sample ready to use. Figure 8 shows an execution example of this package, where JADE GUI, text console, and some instances of the Render Engine can be seen.



```
Manager[12]: Sending notification t
[A1@pandemonium:1099/JADE]: Beginni
Manager[13]: Sending notification t
[E10@pandemonium:1099/JADE]: Beginn
Manager[14]: Sending notification t
[A4@pandemonium:1099/JADE]: Beginni
Manager[15]: Sending notification t
[A2@pandemonium:1099/JADE]: Beginni
Manager[16]: Sending notification t
Manager: Sending Objective notifica
Registry - Service Registered: Ammo

Activating Agent <A3@pandemonium:10
    * Name: Ammo_Allied
    * Type: Ammo_Allied
Registry - Service Registered: Back

Activating Agent <E9@pandemonium:10
    * Name: Backup_Axis
    * Type: Backup_Axis

Activating Agent <E7@pandemonium:10
    * Name: Backup_Axis
    * Type: Backup_Axis
Registry - Service Registered: Back

Activating Agent <A6@pandemonium:10
    * Name: Backup_Allied
    * Type: Backup_Allied

Activating Agent <E6@pandemonium:10
    * Name: Backup_Axis
    * Type: Backup_Axis

Activating Agent <E8@pandemonium:10
    * Name: Backup_Axis
    * Type: Backup_Axis
Registry - Service Registered: Ammo

Activating Agent <E3@pandemonium:1099/JADE - id: 7> ... [OK]
    * Name: Ammo_Axis
    * Type: Ammo_Axis

Activating Agent <E5@pandemonium:1099/JADE - id: 8> ... [OK]
    * Name: Backup_Axis
    * Type: Backup_Axis
Registry - Service Registered: Medic_Axis

Activating Agent <E2@pandemonium:1099/JADE - id: 9> ... [OK]
    * Name: Medic_Axis
    * Type: Medic_Axis
```

Figure 8: Execution example of *JGOMAS* package.

Following there is a more detailed explanation of executing either *JGOMAS* MAS and Render Engine.

4.1 Executing JGOMAS MAS

One of the advantages of *JGOMAS* is its flexibility for configuring its start-up. This is possible because user can choose the number and type of agents, along with the parameters for each agent.

These configuration parameters expected for the agents depends on its type:

- The parameters addressed to the *Agent Manager* are the following:

  1. Number of *Troop* agents to start the match.
  2. Map (scenario) where the match is going to be played.
  3. Graphic viewers refresh frequency (in milliseconds).
  4. Match duration (in minutes).

- *Troop* agents accept just one parameter, the team, that can be either `ALLIED` or `AXIS`.

> User can consider necessary to increment the number of parameters accepted by his agents. For that, he must create his own agents, as it was explained previously, to handle these new parameters.

An example of execution of *JGOMAS* is:

```
java -classpath lib\jade.jar;lib\jadeTools.jar;lib\Base64.jar;
lib\http.jar;lib\iiop.jar;lib\beangenerator.jar;.lib\jgomas.jar;student.jar;.
jade.Boot -gui Manager:es.upv.dsic.gti-ia.jgomas.CManager(4 map_04 125 10)
A1:student.MyMedic(ALLIED) A2:student.MyMedic(ALLIED) E1:student.MyMedic(AXIS)
E2:student.MyMedic(AXIS)
```

According to this example, the *Agent Manager* begins a match with 4 players, which play in the map `map_04` for 10 minutes, at 8 frames per second (125 ms.).

4.2 Executing Render Engine

This application allows to launch an OpenGL window to display *JGOMAS* agents in a 3D world (it is based on the OpenScenGraph toolkit [3]). It must be found in the `\bin\render` directory. Thus, user can observe the course of a match. However, at this point, user cannot use Render Engine to send orders to agents.

User may add some additional parameters when launching this program:

1. $--$ *server*: name or ip of the machine where is launched *Agent Manager* (acting as server). Default value is localhost.
2. $--$ *port*: is the port used to communicate to server (default value is 8001).
3. $--$ *h*: it shows the help of the render engine.

4.2.1 Viewer Window

Once a connection is established, this window is created, where the 3D virtual environment is rendered using OSG.

- Mouse (default camera):
  - *Double-Click over an agent*: shows info about agent clicked on.
  - *Left-Click*: rotates the field.
  - *Center-Click*: strafe view.
  - *Right-Click*: click and roll Y axis to zoom.
- Keyboard:

- **1 -** Select "Trackball" camera manipulator (default)
- **2 -** Select "Flight" camera manipulator
- **3 -** Select "Drive" camera manipulator
- **4 -** Select "Terrain" camera manipulator
- **5 -** Select "UFO" camera manipulator
- **Escape -** Quit the application
- **O -** Save screenshot to “*saved_image ∗ .jpg*”
- **Z -** Stop the recording of a camera path, save it to “saved_animation.path” and reset the camera to the beginning of the animation
- **b -** Toggle backface culling
- **f -** Toggle full-screen rendering
- **h -** Show help
- **l -** Toggle lightning
- **o -** Write scene graph to *"saved_model.osg"*
- **s -** Toggle instrumentation
- **t -** Toggle texturing
- **v -** Toggle block and vsync
- **w -** Cycle through polygon fill modes (fill, wireframe, dots)
- **z -** Start recording camera path
- **“Drive” camera manipulator:**
  - **Down -** Look downwards
  - **Space -** Reset the camera to the home position
  - **Up -** Look upwards
  - **a - Right Mouse Button - Center Mouse Button -** Speed
  - **q - Mouse Y -** Control speed
- **“Flight” camera manipulator:**
  - **Space -** Reset the camera to the home position
  - **a -** No yaw when banked
  - **q -** Automatically yaw when banked
- **“Terrain” camera manipulator:**
  - **Space -** Reset the camera to the home position
  - **+ -** In stereo, increase fusion distance
  - **− -** En stereo, decrease fusion distance
- **“Trackball” camera manipulator:**
  - **Space -** Reset the camera to the home position

    - **+ -** In stereo, increase fusion distance
    - **− -** In stereo, decrease fusion distance
  - **"UFO" camera manipulator:**
    - **H -** Reset the camera to the home position

4.2.2 Viewer Window

Once a connection is established, this window is created, where the 3D virtual environment is rendered under standard OpenGL.

At any moment, the user can select any troop agent in the window (by means of the left mouse button) and check the main features of this agent (live, ammo, position).

Moreover, the following keyboard shortcuts and hotkeys are available while the *JGOMAS* Render Engine is running (OSG standard shortcuts and hotkeys):

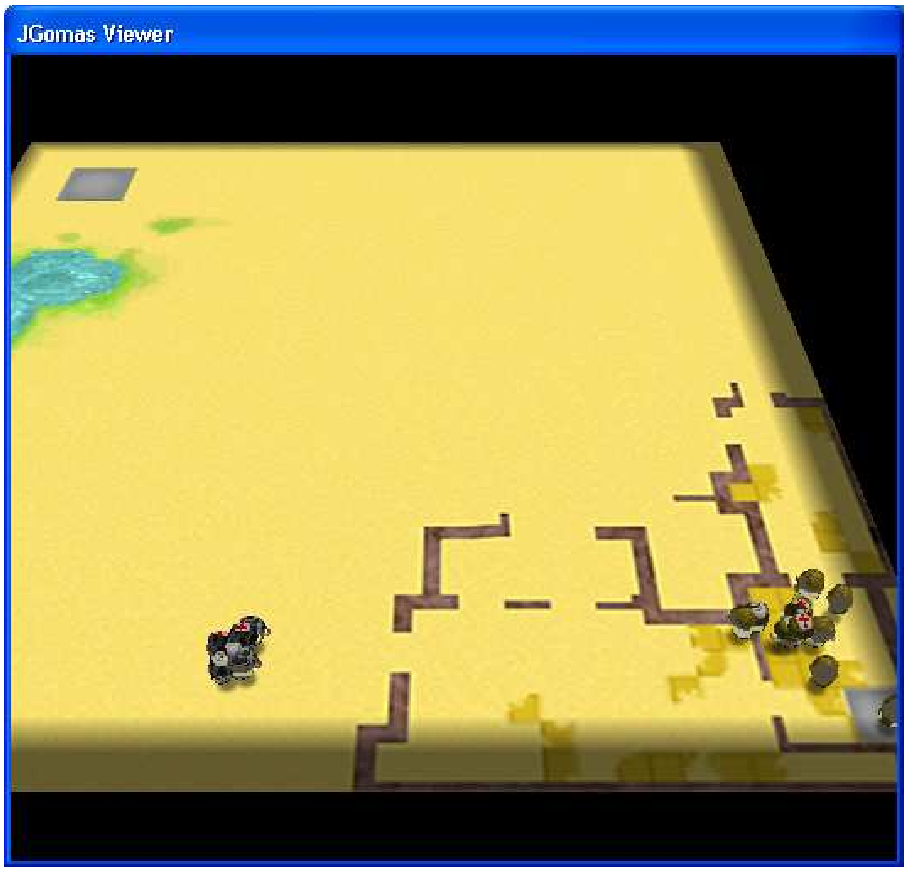


Figure 9: 3D Viewer Window

5 Map Creation

As it has been stated, *JGOMAS* may use different maps for its execution. This maps are stored at the folder `bin\data\maps` of the distribution. In this folder there is one subfolder for each map with the name `map_XX`, where `XX` is the number of this map. This folder contains different files defining the map. Next, we are going to see the contents of the `map_04` to illustrate the configuration of a map:

- *map_04_cost.txt*: This file defines the walls of the map by using `*` to indicate it. The contents of this file is the following:

```
********************************
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                              *
*                         **  **
*                         *    *
*               *         *    *
*               *         *    *
*               *         *    *
*             ********   **    *
*             *      *****     *
*             *          *     *
*   ***  ******          *     *
*             *                *
*                              *
*                        *     *
********************************
```

- *map_04_terrain.bmp*: This file defines the art of the map, as it can be seen in the left image of figure 10.

- *map_04_cost.bmp*: This file defines the walls of the map by using a black & white image, where black represents the wall. It may be seen in the right image of figure 10.

- *map_04.txt*: This file contains the definition of different configuration parameters for the MAS and the Render:

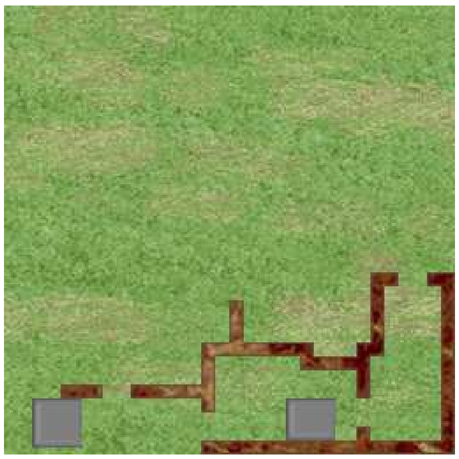
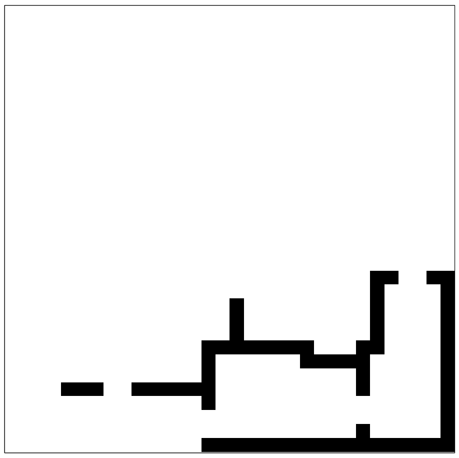



Figure 10: Left image: map_04_terrain.bmp – Right image: map_04_cost.bmp

- JADE_OBJECTIVE: Flag's initial location.
- JADE_SPAWN_ALLIED: Allied base's location.
- JADE_SPAWN_AXIS: Axis base's location.
- JADE_COST_MAP: Size and name of the cost's file.
- RENDER_ART_MAP: Size and name of the art file.
- RENDER_COST_MAP: Size and name of the cost's art file.
- RENDER_HEIGHT_MAP: Size and name of the height's art file.

The contents of such file is the following:

```
[JADE]
JADE_OBJECTIVE: 28 28
JADE_SPAWN_ALLIED: 2 28 4 30
JADE_SPAWN_AXIS: 20 28 22 30
JADE_COST_MAP: 32 32 map_04_cost.txt
[JADE]

[RENDER]
RENDER_ART_MAP: 256 256 map_04_terrain.bmp
RENDER_COST_MAP: 32 32 map_04_cost.bmp
RENDER_HEIGHT_MAP: 32 32 map_04_heightmap.bmp
[RENDER]
```

References

[1] id Software. Checked on January 31, 2006. http://www.idsoftware.com/

[2] Neider, J., Davis, T., Woo, M.: OpenGL Programming Guide. Addison W., 3^{a} Ed, 1999 The Industry's Foundation for High Performance Graphics. Checked on January 31, 2006. http://www.opengl.org/

[3] OpenSceneGraph. Checked on October 25, 2006. http://www.openscenegraph.org/

[4] JADE (Java Agent DEvelopment Framework). Checked on January 31, 2006. http://jade.tilab.com/

[5] JAVA Technology. Checked on January 31, 2006. http://java.sun.com/

[6] The Foundation for Intelligent Physical Agents. Checked on January 31, 2006. http://www.fipa.org/